L9: Entry 1 of 2 File: USPT Aug 22, 2000

DOCUMENT-IDENTIFIER: US 6108744 A
TITLE: Software interrupt mechanism

Brief Summary Text (3):
In a conventional interrupt mechanism which is typically used in non real-time systems such as the UNIX operating system, interrupt processing is provided at a hardware interrupt level. (UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Ltd.) In such an interrupt mechanism, interrupts are typically allocated one of a number of different interrupt levels, for example eight, where 0 is the highest level and 7 is the lowest level. When an interrupt of a given level (say level 4) is being processed, an interrupt of a higher level (say level 2) can pre-empt (interrupt) the processing of the level 4 interrupt, whereby the level 2 interrupt is completed before processing of the level 4 interrupt is completed. A disadvantage of this approach is, however, that when the interrupt of a given level (say level 4) is being processed, lower level interrupts are masked. The masking of lower level interrupts is disadvantageous, particularly in real-time systems, as it prevents real-time processing of the interrupts.

Brief Summary Text (4):
An alternative approach to the handling of interrupts, which finds application to real-time operating systems, employs the use of interrupt threads. An example of a real time operating system which uses interrupt threads to process interrupts is the CHORUS/ClassiX operating system. This operating system is largely written in high level computer language to be hardware independent and comprises the minimum of hardware dependent "glue" code. An interrupt thread having a very short critical section is used to process interrupts rather than performing this at the interrupt level itself. Specifically, an interrupt handler wakes up a high priority thread (an interrupt thread) using a binary semaphore. The interrupt thread which has been activated carries out the necessary tasks and then control can be returned to the interrupted thread. Although this approach is better than a hardware level approach in a real-time environment (as there is no masking of interrupts), it is still not entirely satisfactory due to the delays involved in rescheduling, i.e. in switching from the interrupted thread to the interrupt thread and then back again.

Detailed Description Text (10):
In contradistinction thereto, with a software interrupt mechanism 36, 37, 38 according to an embodiment of the invention, there is no need to wake up an interrupt thread. At an interrupt level, when an interrupt handler has been called and when the supervisor is about to return from interrupt, software interrupts are called according to relative priorities of those software interrupts.

Detailed Description Text (12):
The interrupt mechanism allows interrupt handlers (software interrupt handlers) to be invoked from an interrupt or other dedicated stack in a fully controlled way. In an embodiment of the invention a software interrupt is entirely managed by software and requires no specific hardware mechanism to trigger it. A software interrupt can be considered to have the lowest interrupt priority among all interrupts (i.e. an interrupt priority below all hardware interrupt levels). Thus, any hardware interrupt handler can pre-empt a software interrupt handler. In other words, a software interrupt handler is always called with all interrupts unmasked.

Detailed Description Text (17):
When a software interrupt handler calls a higher priority software interrupt handler, the latter software interrupt handler will be invoked after the completion of the current one. There is no real pre-emption in this case.

Detailed Description Text (19):
While a software interrupt handler of priority 4 is executing, a hardware interrupt occurs which calls a software interrupt handler of priority 3. The hardware interrupt pre-empts immediately the current software interrupt handler. When this hardware interrupt handler completes its processing, the software interrupt handler 3 will be triggered and then, after it completes, the software interrupt handler of priority 4 continues its processing.

Detailed Description Text (25):
Software interrupts can also be pre-emptable. Interrupts are not masked when a software interrupt handler is called so that so it is possible that a (hardware) interrupt occurs and pre-empts the current processing. The hardware interrupt handler can also trigger a software interrupt. More complex schemes are also possible with multiple interrupts and multiple triggered software interrupts. On return from the penultimate nested level of interrupt, if a higher priority software interrupt has been triggered, only higher priority software interrupts are processed as is represented in Table 2 below.

Detailed Description Text (41):
The trigger primitive 94 can be called to trigger a software interrupt handler 98. A software interrupt handler can be triggered after all hardware interrupts have been processed at the interrupt level. Otherwise, it can be triggered immediately.

Detailed Description Text (49):
5) a process management component (processSoftInterrupt()), which provides the generic routine called either by the trigger component, or at the interrupt level at the end of hardware interrupt processing; this component calling all triggered software interrupt handlers beginning with the highest priority and progressing to the lowest priority.

Detailed Description Text (51):
A global variable is shared between the system supervisor and the portable components to indicate that one or multiple software interrupt handlers are ready to be called. This variable is named: SoftIntrToCall.

Detailed Description Text (54):
softPendingList [SOFTINTR.sub.-- MAX.sub.-- PRIO]: This provides a table of doubly linked lists for recording all pending software interrupts. There is one list per priority level. Each time a software interrupt is triggered, it is appended according to its priority in its corresponding list. When a software interrupt handler is called, its corresponding element is unlinked from the list.

Detailed Description Text (82):
The operation of the management component is also illustrated in the flow diagram of FIG. 9. A global variables f.sub.-- imsIntrLevel in step S30 is updated by the supervisor (Component 18 FIGS. 1 and 3) recording the nesting level of hardware interrupt. When there is no interrupt this variable is equal to 0. Upon interrupt, this variable is incremented, and before returning from interrupt it is decremented. For instance when this variable is equal to 2, it means that two hardware interrupts are nested. In other words, a hardware interrupt has preempted another hardware interrupt. Specifically, when the management component is called, this global variable is tested, and when it is strictly greater than 2, the operation ends immediately. This means that another hardware interrupt is pending. When f.sub.-- imsIntrLevel is equal to 2, a software interrupt preemption has occurred, otherwise f.sub.-- imsIntrLevel is equal to 1. Specifically, when the management component is called, softIntrToCall is set to zero and curSoftIntrPrio is saved in a local variable prevSoftIntrPrio in step S32. Then, the management component loops on non empty lists from the highest priority to prevSoftIntrPrio. Each time a software interrupt is unlinked from a list the highest priority is updated in pMapPrio when the list becomes empty. Thus, in step S34, for the current highest priority, control is passed to step S36 which unmasks all interrupts. Then, in step S38, the software interrupt handler is called from the interrupt to be processed. Thus, control is passed at S40 to a conventional interrupt handler.

Detailed Description Text (83):
At step S42, on return of control following interrupt processing, in step S44, all interrupt are masked. Then, in step S46, the highest priority is compared against prevSoftIntrPrio, and control is passed to step S34 if this assertion is true, otherwise control is passed to step

S48.

Detailed Description Paragraph Table (1):

TABLE 1 _____ SOFT INTERRUPT---> 1 Save registers 2 Goto 4 INT i-----> 1 Save registers 2 Switch on interrupt stack 3 Call the interrupt handler----------...>... 4 If any software interrupt call them---------->... 5 Switch on system stack of the interrupted thread 6 Call the scheduler 7 Restore registers <------- 9 Return from interrupt _____

Detailed Description Paragraph Table (2):

TABLE 2 _____ INT i -----> 1 Save registers 2 Switch on interrupt stack 3 Call the interrupt handler---------->... 4 If any software interrupt call them---------->... INT j-------> 1 Save registers 2 Call the interrupt handler---------->... INT x---------->... <---------- 4 If we return from interrupt j and if one or multiple higher priority software interrupt has been triggered call it or them---------->... 5 Restore registers <---------- 6 Return from interrupt j 5 Switch on system stack of the interrupt thread 6 Call the scheduler 7 Restore registers <------- 9 Return from interrupt i _____

Detailed Description Paragraph Table (4):

TABLE 4 _____ if [interrupt level or all interrupts are masked] then if [current software interrupt priority < curSoftIntrPrio] softIntrToCall=1 fi return else call Stack Switch Component fi return _____

Detailed Description Paragraph Table (5):

TABLE 5 _____ f.sub.-- imsIntrLevel < 2 otherwise return Save curSoftIntrPrio in a local variable: prevSoftIntrPrio start: Loop on non empty lists Get the highest priority and update it if list is empty unmask all interrupts call the handler Mask all interrupts while [Highest priority < prevSoftIntrPrio]go to start Restore curSoftIntrPrio from prevSoftIntrPrio Return _____

CLAIMS:

8. The interrupt mechanism of claim 1, comprising an invocation component for invoking a software interrupt handler, wherein each software interrupt handler is allocated a priority level and wherein said management component is configured to be operable to call multiple said software interrupt handlers in order of priority.

16. The operating system of claim 9 comprising an invocation component for invoking a software interrupt handler, wherein each software interrupt handler is allocated a priority level and wherein said management component is configured to be operable to call multiple said software interrupt handlers in order of priority.

Previous Doc Next Doc Go to Doc#

L9: Entry 2 of 2 File: USPT Aug 30, 1988

DOCUMENT-IDENTIFIER: US 4768149 A
TITLE: System for managing a plurality of shared interrupt handlers in a linked-list data structure

Abstract Text (1):
A system is disclosed for managing a plurality of interrupt handlers in a linked-list data structure, for servicing a plurality of input/output devices sharing a common interrupt line in a microcomputer. The system provides for an orderly method to link a newly loaded interrupt handler routine into a linked-list data structure consisting of previously loaded interrupt handler routines. The system further provides for an orderly method to share a common interrupt line among a plurality of input/output devices being serviced by the interrupt handlers. The system further provides for an orderly means to unlink a particular interrupt handler routine from the linked-list data structure when a corresponding input/output device is to be deactivated. The system finds special utility in a multitasking operating system environment where input/output devices can be deactivated in a different sequence from that in which they were originally activated.

Brief Summary Text (8):
FIG. 2 shows the alternate prior art approach employing the interrupt method. In FIG. 2, the CPU 20 is connected by means of the data bus 26 to the I/O devices 27 and 29 and to the RAM 32 and the ROM 30. An additional hardware element included in the system of FIG. 2 is the programmable interrupt controller (PIC) 22 which is also connected to the data bus 26. Each of the I/O devices 27 and 29 has a separate, dedicated interrupt line 28 going to the programmable interrupt controller 22. The programmable interrupt controller 22 receives an interrupt signal (IRQ) from one of the I/O devices 27 or 29 over that one of the interrupt lines 28 dedicated to that I/O device and the PIC 22 outputs an interrupt request (IREQ) to the CPU 20. The CPU is allowed to execute its main program until it receives an interrupt request (IREQ) from the PIC 22. It then stops to service that I/O device issuing the corresponding interrupt signal (IRQ). The interrupt request (IREQ) sent by PIC 22 to the CPU 20 informs the CPU that it should complete whatever instruction that is currently being executed and then fetch a new routine, called an interrupt handler, which will service the I/O device requesting the interrupt. Once this servicing is complete, the CPU 20 will resume its main program from the point where it left off. The interrupt method of FIG. 2 significantly increases system throughput over the status polling method of FIG. 1.

Brief Summary Text (11):
Each I/O device in FIG. 2 generally has a dedicated program which is associated with its specific operational requirements, generally called a service routine or interrupt handler. In the IBM Personal Computer, access is gained to the interrupt handler by means of an interrupt vector. The term "interrupt vector" means a location in memory which is used to hold the address of another location where the interrupt handler program is located. Initially, when system power is turned on, one of the initialization operations is to construct a set of at least eight interrupt vectors in the low order memory locations in the RAM 32. Each interrupt vector represents the memory address for the starting point of an interrupt handler routine for handling the interrupt signal from one of the eight I/O devices requesting an interrupt on a corresponding one of the eight interrupt lines 28. In the prior art mode of operation of the IBM Personal Computer, the Intel 8259A Programmable Interrupt Controller 22 detects an interrupt signal (IRQ0 through IRQ7) when an I/O device 27 or 29 pulls the corresponding one of the interrupt lines 28 to a relatively high potential. If the PIC 22 honors the request, the IREQ output line to the Intel 8088 CPU 20, will go high, indicating to the CPU that a valid interrupt request has been made. When an interrupt request is present at the IREQ line at the CPU 20, the Intel 8088 CPU 20 enters its interrupt acknowledge machine cycle. The interrupt

acknowledge machine cycle completes the processing of the current instruction and stores away current status flags, the contents of certain operating registers, and the current instruction pointer onto a last-in/first-out stack provided in the RAM 32. The Intel 8088 CPU 20 then issues the first of two interrupt acknowledge (IACK) pulses which signal the PIC 22 that the CPU 20 has honored its interrupt request. The Intel 8259A Programmable Interrupt Controller 22 is now ready to initiate the execution of the interrupt handler routine by means of one of the interrupt vectors in the RAM 32. This is done during the sequence of the two interrupt acknowledge pulses (IACK) issued by the CPU 20. The second interrupt acknowledge pulse causes the PIC 22 to place a single interrupt vector byte onto the data bus 26, which pertains to the desired interrupt vector corresponding to the interrupt handler routine. When the CPU 20 receives the interrupt vector byte from the PIC 22, it computes the actual address of the interrupt vector location in the RAM 32. The contents of the addressed interrupt vector in the RAM 32 is then accessed by the CPU 20, that contents being the address of the beginning of the interrupt handler routine which will service the interrupt of the I/O device requesting attention. Program execution of the interrupt handler routine is then carried out by the CPU 20.

Brief Summary Text (23):
It is yet a further object of the invention to manage a plurality of shared interrupt handlers in a linked-list data structure, in an improved manner.

Brief Summary Text (25):
These and other objects, features and advantages of the invention are accomplished by the system disclosed herein. A system is disclosed for managing a plurality of interrupt handlers in a linked-list data structure, for servicing a plurality of input/output devices sharing a common interrupt line in a microcomputer. The system provides for an orderly method to link a newly loaded interrupt handler routine into a linked-list data structure consisting of previously loaded interrupt handler routines. The system further provides for an orderly method to share a common interrupt line among a plurality of input/output devices being serviced by the interrupt handlers. The system further provides for an orderly means to unlink a particular interrupt handler routine from the linked-list data structure when a corresponding input/output device is to be deactivated. The system finds special utility in a multitasking operating system environment where input/output devices can be deactivated in a different sequence from that in which they were originally activated.

Brief Summary Text (26):
In accordance with the invention, an interrupt sharing program provides for an orderly method to link a newly loaded interrupt handler routine into a linked-list data structure or chain of previously loaded interrupt handler routines. The interrupt sharing program further provides an orderly method to share the interrupt line while a given application program is active. Still further, the interrupt sharing program provides an orderly means to unlink a particular interrupt handler routine from a chain of a plurality of interrupt handler routines, when the application program corresponding to the particular interrupt handler routine is to be deactivated in a multitasking operating system environment.

Detailed Description Text (5):
Whenever an application program AP(N) requires the use of the shared interrupt line IRQ7, it must be accompanied by an interrupt sharing program S(N), as shown in FIG. 14. (N is the identity of the application A, B, C, etc.) In accordance with the invention, the interrupt sharing program S(N) consists of three parts, the linking logic routine L(N) shown in FIG. 6, the interrupt handler routine H(N) shown in FIG. 7, and the unlinking logic routine U(N) shown in FIG. 8. The interrupt sharing program S(N) can be loaded when the application program AP(N) is loaded, and it can occupy a contiguous portion of the memory 32 or it can occupy another known position in the memory. FIGS. 6 and 9-12 show how the linking logic L(N) associated with each application program AP(N) requiring the use of the IRQ7 interrupt line, provides for an orderly method to link a newly loaded interrupt handler routine H(N) into a chain of previously loaded interrupt handler routines. FIGS. 5 and 7 show how the interrupt handler H(N) provides an orderly method to share the interrupt line IRQ7 while a given application program AP(N) is active. FIGS. 8 and 13 show how the unlinking logic U(N) provides an orderly means to unlink a particular interrupt handler routine H(N) from a chain of a plurality of interrupt handler routines, when the application program AP(N) corresponding to the particular interrupt handler routine H(N) is to be deactivated in a multitasking operating system environment.

Detailed Description Text (31):

Reference is now made to the linking routine L(N) flow diagram of FIG. 6. Each major step shown in the flow diagram of FIG. 6 contains example assembly language instructions for carrying out the step. When the operating system or alternately the application program AP(N) requires the linking of the interrupt handler H(N) into the chain of interrupt handlers, the operating system or the application program AP(N) passes to the entry point 150 of the flow diagram of FIG. 6. The entry point 150 passes to the first step 152 which carries out the function of disabling the interrupts. Step 152 passes to step 154 which carries out the process of setting the forward pointer F(N) for the interrupt handler H(N), to equal the value of the contents of the interrupt vector V(7). Step 154 then passes to step 156 where a test is conducted to determine if the existing interrupt handler, which had been pointed to by the existing contents of the interrupt vector V(7), was in fact an interrupt return instruction (IRET). If in fact it was an interrupt return instruction, then that existing interrupt handler was the default interrupt handler initially set up by DOS. In this case, the new interrupt handler H(N) now being linked into the chain, will be a first non-default interrupt handler in the chain, and therefore its flag byte must be set to indicate this condition. The compare instruction (CMP) in step 156 is satisfied and therefore the jump-if-not zero (JNZ) instruction in step 156 causes step 156 to pass to step 158 where the flag byte F(N) is set to indicate that the interrupt vector H(N) now being linked into the chain, is the first interrupt vector in the chain. Step 158 then passes to step 160. Alternately, if step 156 determines that the interrupt handler H(N) is not the first in the chain, then step 156 passes directly to step 160. Step 160 loads the address of the entry point for the interrupt handler H(N) into the location of the interrupt vector V(7). Thus, when an interrupt signal occurs on the interrupt line IRQ7, when the CPU 20 accesses the interrupt vector V(7), the first interrupt handler in the chain to be executed will be the most recently linked interrupt handler H(N). In the flow diagram of FIG. 6, step 160 then passes to step 162 where the interrupts for the interrupt level IRQ7 are unmasked. Step 162 then passes to the exit point 164 which returns control to the operating system or alternately to the application program AP(N).

Detailed Description Text (38):
As an optional feature, the flag byte F(A) in the control block C(A) can also be used to preserve the mask bit settings for the programmable interrupt controller 22, as each consecutive interrupt handler H(N) is processed. The Intel 8259A programmable interrupt controller 22, which is described in the above referenced iAPX 86, 88 Users Manual, includes an interrupt mask register. Rather than all eight interrupts IRQ0 through IRQ7 being disabled or enabled at the same time, the interrupt mask register allows individual interrupt line masking. The interrupt mask register in the programmable interrupt controller 22, is an eight bit register with bits 0-7 directly corresponding to the interrupt lines IRQ0 through IRQ7. Any of these eight interrupt lines can have their input masked by writing into the interrupt mask register and setting the appropriate bit. Likewise, any of the eight interrupt lines can have its input to the programmable interrupt controller 22 enabled by clearing the correct interrupt mask register bit. The utility of this feature is, for example, when a particular interrupt handler routine H(N) is to only be interrupted by specific lower priority interrupts and certain higher priority interrupts are to be disabled during its service routine. The flag byte F(A) in the control block C(A) contains the eight masking bits which are loaded into the interrupt mask register of the programmable interrupt controller 22. The flag byte F(A) in the control block C(A) serves two purposes in this regard. First, it serves as a log of the setting of the mask bits in the programmable interrupt controller 22 when the linking routine L(A) installed the interrupt handler H(A). Secondly, the flag byte F(A) serves specific purpose during normal interrupt handling operations. In the process previously described for searching the chain for the interrupt handler corresponding to the device issuing an interrupt, when a given interrupt handler determines that its corresponding I/O device did not cause the current interrupt, it will pass control to the next interrupt handler in the chain. However, an exception to this passage of control to the next interrupt handler will occur when the flag byte F(N) indicates that the next interrupt handler on the chain would not have normally received control when this particular interrupt signal occurred on the IRQ7 line. In such a situation, normally, the next interrupt handler would not even have been accessed by the interrupt vector V(7) because the issuance of an interrupt request IREQ from the programmable interrupt controller 22 to the CPU 20 would have been masked by the mask bit in the programmable interrupt controller 22. Therefore, in such a situation, the current interrupt handler H(N) having control will not pass that control onto the next interrupt handler in the chain H(N-1), but instead, can either pass control back to the main program which was interrupted by the current interrupt, or alternately it can pass control to the second next interrupt handler H(N-2) by reading the contents of the pointer P(N-1) of the next interrupt handler H(N-1) and then skipping the next interrupt handler H(N-1) so as to pass control to the

second next interrupt handler H(N-2).

Detailed Description Text (39):
The flag byte F(A) also has some significance in the unchaining process. As was previously mentioned, and as will be further explained below, when an interrupt handler H(N) is taken out of the chain, the contents of its pointer P(N) is loaded into the pointer P(N+1) of the control block C(N+1) of the previous interrupt handler H(N+1). The unlinking logic U(N) also loads the contents of the flag byte F(N) into the flag byte field F(N+1) of the preceding interrupt handler H(N+1) control block C(N+1). This will enable the preservation of the mask byte for the programmable interrupt controller 22.

Detailed Description Paragraph Table (2):
TABLE II LINKING
CODE EXAMPLE PUSH ES
CLI ; Disable interrupts ; Set forward pointer to value of interrupt vector in low memory ASSUME CS:CODESEG,DS:CODESEG PUSH ES MOV AX,350FH ; DOS get interrupt vector INV 21H MOV SI,OFFSET CS:FPTR ; Get offset of your forward pointer ; in an indexable register MOV CS: [SI],BX ; Store the old interrupt vector MOV CS:[SI+2],FS ; in your forward pointer for chaining CMP CMP BYTE PTR ES: [BX], 0CFH ; Test for IRET JNZ SETVECTR MOV CS:FLAGS,FIRST ; Set up first in chain flag SETVECTR: POP ES PUSH DS ; Make interrupt vector in low memory point to your handler MOV DX,OFFSET ENTRY ; Make interrupt vector point to your ; handler MOV AX,SEG ENTRY ; If DS not = CS, get it MOV DS,AX ; and put it in DS MOV AX,250FH ; DOS set interrupt vector INT 21H POP DS Unmask (enable) interrupts for your level IN AL,IMR ; Read interrupt mask register JMP $+2 ; IO delay AND AL,07FH ; Unmask interrupt level 7 OUT IMR,AL ; Write new interrupt mask MOV AL,SPC -- EOI ; Issue specific EOI for level 7 JMP $+2 ; IO delay OUT OCR,AL ; to allow pending level 7 interrupts ; (if any) to be serviced STI ; Enable interrupts POP ES

CLAIMS:

1. In a microcomputer including a central computing means, a memory and a plurality of I/O devices connected together by a bus, each of said I/O devices having an output connected to a common interrupt line, for transmitting a corresponding interrupt signal on said interrupt line, and each of said I/O devices having an input connected to said common interrupt line, for monitoring said interrupt line for an occurrence of an interrupt signal thereon and for blocking a transmission of subsequently occurring interrupt signals in response thereto, a system for enabling said I/O devices to share said common interrupt line, comprising:

a plurality of interrupt handler rountines stored as an ordered chain in said memory, each having an instruction portion containing an ordered sequence of instructions executable by said central computing means, to service interrupt demands of a respective one of said I/O device, and each interrupt handler routine having a control block portion located at a fixed relative position from a beginning instruction position of said instruction portion, for storing a pointer address to a beginning instruction position of the instruction portion for a next occuring interrupt handler routine in said chain;

said central computing means interrupting an existing execution of a main program and accessing over said bus, an interrupt vector location in said memory in response to an interrupt signal on said interrupt line, said vector location storing the address of a beginning instruction position of the instruction portion for a first occurring interrupt handler routine in said chain corresponding to a first I/O device of said plurality of I/O devices;

said central computing means executing a status determination segment of said instruction portion of said first interrupt handler routine and in response thereto, accessing over said bus, an interrupt status value stored by said first I/O device, and determining whether said first I/O device caused said interrupt signal to be transmitted on said interrupt line;

said central computing means executing a service routine segment of said instruction portion of said first interrupt handler routine in response to a determination by said central computing means that said first I/O device caused said interrupt signal, for servicing an interrupt demand of said first I/O device;

said central computing means executing a global rearm segment of said instruction portion of said first interrupt handler routine and in response thereto, transmitting a global rearm

message over said bus to all of said plurality of I/O devices, for unblocking a transmission of said subsequently occurring interrupt signals from said I/O devices, prior to returning control back to said main program;

said central computing means excuting a control transfer segment of said instruction portion of said first interrupt handler rountine in response to a determination by said central computing means that said first I/O device did not cause said interrupt signal, said central computing means in response thereto, accessing over said bus, said pointer address in said control block portion of said first interrupt handler routine, for commencing execution of the instruction portion of said next occurring interrupt handler in said chain to service interrupt demands in a corresponding second one of said I/O devices;

whereby a plurality of I/O devices can share said common interrupt line.

5. In a microcomputer including a central computer means, a memory and a plurality of I/O devices connected together by a bus, a combination for enabling said plurality of I/O devices to share a common interrupt line connected to said central computing means, comprising:

a plurality of interrupt logic means, each having a first input connected to an interrupt output of a corresponding one of said plurality of I/O devices, and each having an output connected to said common interrupt line, for transmitting a corresponding interrupt signal on said common interrupt line in response to an interrupt demand output from said corresponding I/O device and storing an interrupt status value;

said interupt logic means each having a second input connected to said common interrupt line, for monitoring said common interrupt line for an occurrence of an interrupt signal thereon and for blocking a transmission of subsequently occurring interrupt signals in response thereto;

a plurality of interrupt handler rountines stored as a chain in said memory, each having an instruction portion containing instructions executable by said central computing means, to service interrupt demands of a respective one of said I/O devices, and each interrupt handler routine having a control block portion located at a fixed relative position from a beginning instruction position of said isntruction portion, for storing a pointer address to said beginning of the instruction portion for a next occurring interrupt handler routine in said chain;

said central computing means interrupting an existing execution of a main program and accessing over said bus, an interrupt vector location in said memory in response to an interrupt signal on said common interrupt line, said vector location storing the address of a beginning instruction position of the instruction portion for a first occurring interrupt handler routine in said chain corresponding to a first I/O device of said plurality of I/O devices;

said central computing means executing a status determination segment of said instruction portion of said first occurring interrupt handler routine and in response thereto, accessing over said bus, an interrupt status value stored by an interrupt logic means for said first I/O device, and determining whether said first I/O device caused said interrupt signal to be transmitted on said common interrupt line;

said central computing means exeucting a service routine segment of said instruction portion of said first occurring interrupt handler routine in response to a determination by said central computing means that said first I/O device caused said interrupt signal, for servicing said interrupt demand of said first I/O device;

said central computing means executing a global rearm segment of said instruction portion of said first occurring interrupt handler routine and in response thereto, transmitting a global rearm message over said bus to said interrupt logic means for all of said plurality of I/O devices, for unblocking a transmission of subsequently occurring interrupt signals from said I/O devices, prior to returning control back to said main program;

said central computing means executing a control transfer segment of said instruction portion of said first occurring interrupt handler routine in response to a determination by said cnetral computing means that said first I/O device did not cause said interrupt signal, said central computing means in response thereto, accessing over said bus, said pointer address in said control block portion of said first occurring interrupt handler routine, for commencing

execution of the instruction portion of said next occurring interrupt handler in said chain to service interrupt demands in a corresponding second one of said I/O devices;

whereby a plurality of I/O devices can share said common interrupt line.

9. In a microcomputer including a central computing means, a memory and a plurality of I/O devices connected together by a bus, each of said I/O devices having an output connected to a common interrupt line, for transmitting a corresponding interrupt signal on said interrupt line, and each of said I/O devices having an input connected to said common interrupt line, for monitoring said interrupt line for an occurrence of an interrupt signal thereon and for blocking a transmission of subsequently occuring interrupt singals in response thereto, a method for enabling said I/O devices to share said common interrupt line, comprising:

storing a plurality of interrupt handler rountines as an ordered chain in said memory, each having an instruction portion containing an ordered sequence of instructions executable by said central computing means, to service interrupt demands of respective one of said I/O devices, and each interrupt handler routine having a control block portion located at a fixed relative position from a beginning instruction position of said instruction portion, for storing a pointer address to a beginning instruction position of the instruction portion for a next occurring interrupt handler routine in said chain;

interrupting said central computing means during an existing of a main program and accessing over said bus, an interrupt vector location in said memory in response to an interrupt signal on said interrupt line, said vector location storing the address of a beginning instruction position of the instruction portion for a first occurring interrupt handler routine in said chain corresponding to a first I/O device of said plurality of I/O devices;

executing in said central computing means, a status determination segment of said instruction portion of said first interrupt handler routine and in response thereto, accessing over said bus, an interrupt status value stored by said first I/O device, and determining whether said first I/O device caused said interrupt signal to be transmitted on said interrupt line;

executing in said central computering means, a service routine segment of said instruction portion of said first interrupt handler routine in response to a determination by said central computing means that said first I/O device caused said interrupt signal for servicing an interrupt demand of said first I/O device;

executing in said central computing means a global rearm segment of said instruction portion of said first interrupt handler routine and in response thereto, transmitting a global rearm message over said bus to all of said plurality of of I/O devices, for unblocking a transmission of said subsequently occurring interrupt signals from said I/O devices, prior to returning control back to said main program;

executing in said central computing means, a control transfer segment of said instruction portion of said first interrupt handler routine in response to a determination by said central computing means that said first I/O device did not cause said interrupt signal, said central computing means in response thereto, accessing over said bus, said pointer address in said control block portion of said first interrupt handler routine, for commencing execution of the instruction portion of said next occurring interrupt handler in said chain too service interrupt demands in a corresponding second one of said I/O devices;

whereby a plurality of I/O devices can share said common interrupt line.

13. In a microcomputer including a central computing means, a memory and a plurality of I/O devices connected together by a bus, each of said I/O devices including an output connected to a common interrupt line, for transmitting a corresponding interrupt signal on said interrupt line, and each of said I/O devices having an input connected to said common interrupt line, for monitoring said interrupt line for an occurrence of an interrupt signal thereon and for blocking a transmission of subsequently occurring interrupt signals in response thereto, a system for enabling said I/O devices to share said common interrupt line, comprising:

a plurality of interrupt handler routines stored as a chain in said memory, to service interrupt demands of respective ones of said I/O devices, and each storing a pointer address to a beginning of a next occurring interrupt handler routine in said chain;

said central computing means interrupting an existing execution of a main program and accessing over said bus, an interrupt vector location in said memory in response to an interrupt signal on said interrupt line, said vector location storing the address of a beginning of a first occurring interrupt handler routine in said chain corresponding to a first one of said I/O devices;

said central computing means executing a status determination routine for determining whether said first one of said I/O devices caused said interrupt signal to be transmitted on said interrupt line;

said central computing means executing a service routine for servicing an interrupt demand of said first one of said I/O devices;

said central computing means executing a rearm routine for unlocking a transmission of said subsequently occurring interrupt signals from said I/O devices, prior to returning control back to said main program;

said central computing means executing a control transfer routine for commencing an execution of a next occurring interrupt handler in said chain to service interrupt demands in a corresponding second one of said I/O devices, if said first one of said I/O devices did not cause said interrupt signal;

whereby a plurality of I/O devices can share said common interrupt line.

14. In a microcomputer including a central computing means, a memory and a plurality of I/O devices connected together by a bus, each of said I/O devices having an output connected to a common interrupt line, for transmitting a corresponding interrupt signal on said interrupt line, and each of said I/O devices having an input connected to said common interrupt line, for monitoring said interrupt line for an occurrence of an interrupt signal thereon and for blocking a transmission of subsequently occurring interrupt signals in response thereto, a method for enabling said I/O devices to share said common interrupt line, comprising:

storing a plurality of interrupt handler routines as a chain in said memory, to service interrupt demands of respective ones of said I/O devices, and each storing a pointer address to a beginning of a next occurring interrupt handler routine in said chain;

interrupting said central computing means during an existing execution of a main program and accessing over said bus, an interrupt vector location in said memory in response to an interrupt signal on said interrupt line, said vector location storing the address of a beginning of a first occurring interrupt handler routine in said chain corresponding to a first one of said I/O devices;

executing in said central computing means, a status determination routine for determining whether said first one of said I/O devices caused said interrupt signal to be transmitted on said interrupt line;

executing in said central computing means, a service routine for servicing an interrupt demand of said first one of said I/O devices;

executing in said central computing means a rearm routine for unblocking a transmission of said subsequently occurring interrupt signals from said I/O devices, prior to returning control back to said main program;

executing in said central computing means, a control transfer routine for commencing an execution of a next occurring interrupt handler in said chain to service interrupt demands in a corresponding second one of said I/O devices, if said first one of said I/O devices did not cause said interrupt signal;

whereby a plurality of I/O devices can share said common interrupt line.

Previous Doc Next Doc Go to Doc#